Document Integrity through Mediated Interfaces[1]

Marcelo Tallis and Robert Balzer
Teknowledge
mtallis@teknowledge.com and bbalzer@teknowledge.com

Abstract

End-to-end integrity for documents is provided by wrapping the tools that manipulate those documents and mediating their operation to cryptographically integrity mark those documents as they are being saved, to check those cryptographic integrity marks as those documents are loaded, and to record an application-level history of the changes to the document. Corrupted documents (those failing to match their cryptographic integrity mark) are automatically repaired by replaying the recorded history of the application-level changes to the document. That recorded history is also used to identify all modifications (including date and author) to any selected portion of the document.

A Document Integrity Manager embodying these capabilities has been developed for Microsoft Word.

1. Introduction

Operating systems provide a variety of facilities for invoking programs and applying them to selected documents. Over time, a particular document is the product of manipulations and transformations carried out by several users and possibly different programs. However, no record of this transaction history of which users and programs have transformed the document is collected and kept. Operating systems only limit, through access control lists, which files users can access and with what file-level rights (read-only, write, and/or execute). No limits are placed on which programs can manipulate a particular document or how it can be transformed. Malicious programs, operating under an authorized user's rights, can corrupt a document directly or by directing another program to modify the document. Malicious users can change the document without any record of those modifications being kept. Under these circumstances, no assurances can be given as to the integrity of the resulting document.

We are remedying this situation by creating an Integrity Manager that monitors and records the tools (i.e. programs), and operations within those tools, being applied to documents to provide an end-to-end audit record of all the transformations performed on those documents. This operation level audit record can be used off-line for attribution (who made a specific change and when did it occur) and on-line for authorization (who and/or which tools are allowed to make particular types of changes to document). This transaction history is also being used to repair corrupted documents by rebuilding them by replaying the recorded sequence of modifications to the document.

2. Technical Approach

Many modern COTS products have extensive APIs that allow most or all of the functionality available in those products interactively through their user interface to also be available to programs running within or external to that product through a "scripting" interface. Some of these programs operate asynchronously from the user, performing background or batch operations, but others operate synchronously with the user automating and expanding their current actions.

Our research focus is on the latter. These synchronous scripting programs need to be invoked whenever the user actions they are expanding or automating occur. This can result from explicit invocation (because the user clicks on a button invoking the "macro" defined by that button) or from an event signaled by the COTS product that triggers the invocation of the scripting program. Although some COTS products support a large number of such events, many only support a few or none at all, and no product supports events for all user actions, especially since a synchronous scripting program may need to be invoked either before or after a particular action.

Our research is aimed at augmenting the scriptablity of COTS products by expanding – beyond the initial set provided by the COTS vendor – the set of user actions and program behaviors that cause events that could trigger scripting programs.

We do this by wrapping [1] the COTS product and monitoring its behavior. The wrapper has access to all of

[1] This work was funded by DARPA's Intrusion Tolerant Systems program under contract F30602-99-1-0542. The views presented are the authors and do not represent the views of DARPA or Rome Laboratories.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 074-0188*

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** 7/14/2001	**3. REPORT TYPE AND DATES COVERED** Research Paper 7/14/2001

4. TITLE AND SUBTITLE Document Integrity through Mediated Interfaces	**5. FUNDING NUMBERS**
6. AUTHOR(S) Tallis, Marcelo; Balzer, Robert	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) DARPA	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) IATAC 3190 Fairview Park Drive Falls Church, VA 22042	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; Distribution unlimited	**12b. DISTRIBUTION CODE** A

13. ABSTRACT *(Maximum 200 Words)*

End-to-end integrity for documents is provided by wrapping the tools that manipulate those documents and mediating their operation to cryptographically integrity mark those documents as they are being saved, to check those cryptographic integrity marks as those documents are loaded, and to record an application-level history of the changes to the document. Corrupted documents (those failing to match their cryptographic integrity mark) are automatically repaired by replaying the recorded history of the application-level changes to the document. That recorded history is also used to identify all modifications

14. SUBJECT TERMS IATAC Collection, information assurance, data integrity	**15. NUMBER OF PAGES** 8
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UNLIMITED

the inter-module interactions within the COTS product and between it and the services it relies upon (i.e. the operating system and middleware such as CORBA and WinSock). By monitoring the resources it utilizes and its interactions with the user through the user interface (just another service it relies upon), the wrapper can detect indicators that suggest particular user actions or program behaviors. By combining these indicators with an examination of the internal state of the COTS products (through its scripting API) the wrapper can determine whether or not the user action or program behavior of interest is occurring, and if so issue a corresponding pseudo-event to trigger the invocation of the appropriate scripting program.

2.1 Prior Pseudo-Event Augmentation of COTS Products

This line of research started with our extension of Microsoft PowerPoint into a domain-specific Design Editor [2] with user supplied semantics for the diagrams. As the user modifies a design, domain specific analyzers monitor the changes being made and feedback the results of their analysis on the diagram within PowerPoint. In that effort we only had to create four psuedo-events beyond those natively provided by PowerPoint: object creation, object destruction, connecting an object to a connector, and breaking a connection between an object and a connector. These psuedo-events allowed our synchronous scripting program within PowerPoint to monitor all the topological changes being made by the user to a diagram and to communicate (via DCOM) those changes to the domain-specific analyzers operating outside PowerPoint.

3. Document Integrity Manager

For this effort, we chose Microsoft Word as the COTS product to wrap and expand into a Document Integrity Manger. We did so because of its large market share and the military's heavy reliance on it for planning and operations (MS Word and PowerPoint are reportedly the two most widely used applications within the military). We also did so because the Word scripting interface was quite comprehensive and well documented and because it supported a reasonable set of native events.

3.1 Technical Challenges

There were four technical challenges in this project. By far, the most challenging was capturing all of the changes being made to a document so that if it were corrupted the recorded history of those changes could be used to rebuild the document. The second was replaying that change history to rebuild the document. The third was detecting whether or not a document had been corrupted. The final technical challenge was using the recorded change history to determine who made particular changes to the document and when they did so.

3.1.1 Capturing All Document Changes

Rebuilding corrupted documents from a recorded history of changes required the capturing of all changes to the document. This was in marked contrast to our prior work on the PowerPoint Design Editor where only four topological operations had to be captured. Our effort here was guided by the following design goals:

- **The history of changes should be described at the application level of the COTS product (MS Word) rather than at the low-level GUI interactions level**. For example, the user action of clicking the command bar *Bold* button should be described as *Toggling Bold face* and not *clicking left mouse button at position (x,y)*. While recording the low-level GUI interactions would be simple – because the Windows operating system already provides a journaling mechanism for capturing these low-level GUI interactions – doing so would make the history much larger and preclude its use for attribution.
- **Changes should be described independently from the particular mechanism in which those changes were performed**. The same action can be performed by several mechanisms. For example, the *Toggle Bold face* action might be invoked by at least three different mechanisms: clicking on a command bar button, selecting it from a menu, and through a keyboard shortcut. These alternative invocation methods are semantically equivalent and representing them as different actions would only add unnecessary complications.
- **The history of changes shouldn't include actions that don't modify the document.** Viewing actions like browsing the document or changing the zoom control don't affect the content of the document and hence should be filtered out of the change history.

The user's actions are determined by combining information from several sources including events reported by the Word and Office API, pseudo-events reported by the wrapper, comparing the content of the document before and after an action, and querying the state of the GUI.

3.1.1.1 Capture Capabilities provided by the Word API

Word's COM interface provides the following functionality that was instrumental in capturing the user's actions:

- An object model that allows the state of a document to be examined.
- Trappable events that report coarse grained operations on documents (e.g., Open, Save, Close, New, Activate window, and Deactivate window). For some of these operations the corresponding event is triggered before the operation is invoked and for others the event is triggered after the operation is completed. For example, for the *Save* and *Close* operations the corresponding event is triggered before the operation is invoked. However, for the *Open* and *New* operations the corresponding event is triggered after the operation is completed.
- Trappable events that report the activation of standard command bar controls like standard *Pushbuttons*, *Combo boxes*, and *Menu items*. For the *Pushbuttons* and the *Menu items* the corresponding event is triggered before the command associated with that button or menu item is executed. For the *Combo boxes* the corresponding event is triggered after the command associated with that combo box is executed.
- A trappable event that is triggered each time that a different region of a document is selected.

Although these capabilities were useful, they were not sufficient for capturing user actions. One reason is that some of the actions that are preceded by a trigger event, such as *Save* or *Close,* may not actually occur because the user aborts the operation when presented with the confirmation dialog box. Word's COM interface doesn't report whether such operations actually occurred or were aborted. This lack of confirmation for operations with preceding events is clearly inadequate for reliably capturing user actions.

A second limitation is that many of the user actions we needed to capture (such as typed text or manipulation of non-standard command bar controls) had neither a preceding nor a following event associated with them, and hence were invisible through the native Word API.

3.1.1.2 Augmenting the Word API with Pseudo-Events.

We therefore constructed our wrapper to generate the following pseudo-events:

- **Completion of the command associated with a pushbutton or a menu item.** This compliments the activation event provided by Word's API and allows the triggered script to examine the state of the document after the execution of the command (e.g., to determine the values entered by the user in the fields of a dialog box displayed by a command).
- **Manipulation of non-standard command bar controls** (e.g., the Undo control)
- **Typed text**
- **Keyboard shortcuts**
- **Dragging text** (e.g., to move or copy text)
- **Completion of the Save and Close operations**. This pseudo-event confirms that the user did not abort the Save or Close operation. For the Save operation it also provides access to the pathname of the saved document and allows the checksum of the saved document to be computed.

3.1.1.3 Document Modification Capture Examples

In this section we describe how a few Word operations were captured. These examples illustrate how information from different sources is combined to identify the actions performed by the user. These examples are presented in order of increasing complexity.

3.1.1.3.1 Setting the Current Selection

This example illustrates a simple case in which the Word API directly reports the operation to be captured.

Most Word operations act upon the current S*election*. The current *Selection* can be a contiguous range within the document that the user has highlighted or can be collapsed into an *Insertion Point*. The user sets the current *Selection* by a variety of methods including: the directional arrows (left, right, up, and down) to move the insertion point, mouse clicks to set the *Insertion Point* at a determined position, dragging the mouse pointer to select a range, and expanding the current selection with Shift+arrow keys.

However, not all of these *Selection* changes need to be recorded. Those that are followed by another *Selection* change without an intervening document modification (e.g., when the user moves the *Insertion Point* several positions ahead by repeatedly pressing the arrow key) are superfluous and aren't recorded.

Capture Procedure

1. The Word API issues a *WindowSelectionChange* event and passes a reference to the Word *Selection* object each time that a different *Selection* is made.
2. The position of the current *Selection* is cached but not recorded until a subsequent document modification is detected.

3.1.1.3.2 Command Bar *Bold* button (B)

This example illustrates a case in which the MS Office API reports a GUI event that can be directly related to the operation to be recorded.

The Command Bar Bold button toggles the *Bold* property of the current *Selection*.

Capture Procedure

1. When a command bar button is pushed the Word API issues a *Click* event and passes a handle to the object that corresponds to that button. The event is triggered **before** the action associated with that button is executed.
2. One property of the Button object is its *Control ID*, a unique and fixed value assigned to each control. This ID is examined to detect the invocation of the *Toggle Boldface* command.

3.1.1.3.3 Command Bar *Style* combo box (Normal)

This example illustrates a case in which the document state resulting after a command has been completed must be examined to determine the parameters of an operation.

The Command Bar Style combo box sets the style of the current *Selection*.

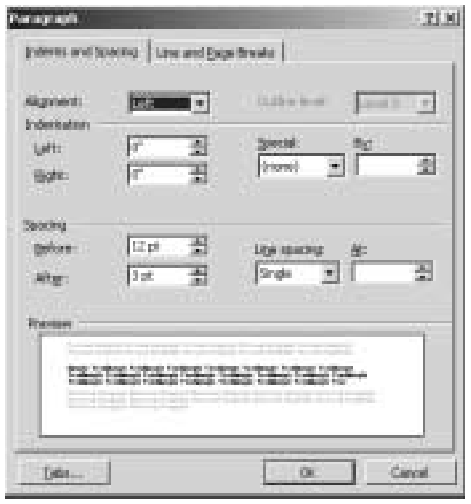
Capture Procedure

1. When the user changes the selection in a command bar combo box, the Word API issues a *Change* event and passes a handle to the Combo Box object as an argument. This event is triggered **after** the command associated with that control is executed.
2. The combo box's ID is used to identify the *Style* command. The style that the user selected is obtained from the *Style* property of the selected text (notice that this reflects the choice made by the user because the combo box Change event is triggered after the associated command is executed). A *ChangeStyle* operation is recorded with the name of the style as its parameter.

3.1.1.3.4 *Format Paragraph* dialog box (Paragraph...)

This example illustrates a case in which the document state before and after the execution of a command must be compared to determine the parameters of an operation. It also illustrates the use of a mediator to intercept the end of the processing of a command invoked through a menu.

The *Format* Paragraph menu item opens a dialog box for setting some paragraph properties of the current Selection (See Figure 1.).



Alignment: Left
Left: 0"
Right: 0"
Special: (none)
By:
Before: 12 pt
After: 3 pt
Line spacing: Single
At:

Figure 1: Format Paragraph dialog box

Capture Procedure

1. When the user selects the *Format Paragraph* menu item the Word API issues a *Click* event and passes a handle to the Menu Item object as an argument. This event is triggered **before** the action associated with that button is executed.
2. The menu item ID identifies the *Format Paragraph* command. The default values for the fields of the Format Paragraph dialog box are cached (this information reflects the state of the document before the execution of the command).
3. A mediator installed in the window message handler for the Menu Bar Pop-up window issues a pseudo-event signaling the **completion** of the procedure that processes this pop-up window. This event necessarily occurs after the *Format Paragraph* command has completed and is used as a surrogate for that event.
4. The pseudo-event triggers code that obtains the new default values for the fields of the *Format Paragraph* dialog box (which now reflect the state of the document after the execution of the command) and compares them with the cached values to determine which fields have changed. It records a *FormatParagraph* operation with those fields and their new values as parameters.

3.1.1.3.5 *Copy/Move* through direct manipulation

This example illustrates a case in which the Word API doesn't provide events that enable these actions to be captured. Hence we mediated the low level GUI interfaces to generate the required pseudo-events. To do so we took advantage of a visual clue in the GUI – that the mouse pointer changes its shape when it is over the selected text

– to determine the nature of the application-level operation in progress.

In a direct manipulation Copy/*Move,* the selected text is dragged with the mouse to another location. If the *Ctrl* key is held down during this operation then it is a *Copy,* otherwise it is a *Move*.

Capture Procedure

1. A mediator installed in the window's message handler for the main Word window reports the **beginning** of the procedure that handles the *Mouse Left-Button Push-Down*.
2. The shape of the mouse pointer at this time is used to determine whether the mouse pointer was over selected text. If so and the left mouse button is depressed then a *drag* operation has been initiated. The current location of the selected text is cached (this location corresponds to the state of the document **before** the *drag* operation).
3. The same mediator mentioned in step 1 reports the **completion** of the window message handler procedure, indicating the completion of the application-level *Copy*/*Move* operation.
4. Several elements of the current state of the document are examined to gather all the information needed to record the *Copy* or *Move* operation. First the state of the *Ctrl* key is checked to distinguish between *Copy* and *Move* operations. Then the now current location of the *Selection* is obtained (it now corresponds to the state of the document after the operation was carried out). Because Word has an *AutoCorrect* feature that automatically inserts or removes extra white spaces around the copied/moved text for the user and because this feature is only active for *Copy/Move* operations performed through the GUI (as opposed to those performed through the scripting API), the effects of this *AutoCorrect* operation must also be recorded for the current *Selection* and, in the case of the *Move,* in the place where the moved text used to be. Finally, the *Copy* or *Move* operation is recorded with parameters indicating the distance (positive or negative) that the selected text was moved and the number of surrounding white spaces added or deleted.

3.1.1.3.6 Typing (and *AutoCorrect*)

This example illustrates another case in which the Word API doesn't provide events that enable these actions to be captured. However, in this case there were a number of extra challenges. The first was that at the time that our mediators reported the typing activity, Word's internal state hadn't been updated so we couldn't use its API to capture the typed text. The second challenge was that capturing the typed text was only part of the job. By some obscure mechanism, Word divides long sequences of typing into undoable chunks (i.e., each *undo* operation undoes exactly one of these chunks of text). We therefore needed to detect exactly when Word chunked the typed sequence so that our recorded sequence of application-level changes would remain synchronized with Word's sequence of undoable units. We were able to overcome these problems through a discovery made after lengthy spying. We observed that Word redirected a keyboard message to a window message handler procedure only when it regarded that typed character as the first character of a new undoable unit. This gave us a way to detect the creation of those typed chunks. We also observed that when Word processed that first character of an undoable unit, its internal state had been updated to include the text typed in the previous chunk. This gave us a way to capture the typed text in each chunk.

Capture Procedure

1. A mediator installed in the window message handler for the main Word window reports the **beginning** of the procedure that handles the *Key-Down* messages. This event signals the beginning of a typing chunk.
2. The previous typing chunk, if any, is obtained through the Word API and recorded.

These two steps are repeated until a different type of event is received. This indicates that the last typing chunk is completed, and that last typing chunk is obtained through the Word API and recorded.

3.1.2 Replaying the Recorded Change History

This section describes the challenges faced in replaying the recorded change history to rebuild a corrupted document.

For most operations this replay is straightforward because the Word API has an operation that corresponds to the recorded change and the history includes all the information required for performing that operation.

However, for some operations (e.g. *AutoCorrect*) there is no corresponding operation in the Word API. For these operations the recorded change either had to be translated into a sequence of available operations, or the captured operation had to be simulated through the GUI so that Word would process it (performing the COM unavailable operation) as if it had come from the user. Sometimes this simulation through the GUI required execution of functions at the OS level to emulate device operations (e.g., keyboard input).

3.1.2.1 Replay Capabilities provided by the Word API

Word's COM interface provides the following functionality that facilitated execution of the captured operations (used for reconstructing a corrupted document).

- Methods to perform application level commands (e.g., Execute the *Toggle Bold Face* command on the selected text)
- Methods that perform fine grained changes to the elements that represent the state of the document (e.g., set the *boldface* property of the text between the positions 1 and 10 equal to True).

3.1.2.2 Document Modification Replay Examples

In this section we describe how the example document modification operations (described above in the Capture section) are replayed. The first three of these examples (Selection change, Command Bar Style combo box, and Format Paragraph dialog box) are skipped because the Word API contains an operation corresponding to these changes and replaying them merely consists of invoking those operations with the recorded parameters.

3.1.2.2.1 *Copy/Move* through direct manipulation Replay

This example illustrates a case in which the Word API does not provide a specific method for replicating this operation and hence it had to be implemented through a sequence of lower level commands. This forced us to create a special mechanism to handle the *undo* of these multiple-step operations atomically.

Although Word does not provide a specific method for *Copy* or *Move* they can easily be implemented through a sequence of lower level Word operations (copying, cutting, pasting, and inserting and deleting white spaces around the copied or moved text).

However, this multi-step implementation breaks the correspondence we've been maintaining between the recorded history and Word's undoable units. What was originally an atomic operation has, in the reconstructed copy, become a sequence of operations. If the recorded history contained an *Undo* of this operation, that *Undo* would operate differently in this reconstructed copy than it did in the original document (in the original document it would have undone the entire *Move* or *Copy* operation, while in the reconstructed copy it would only undo the very last operation in the multi-step sequence).

We handled this problem by placing special *multi-step-begin* and *multi-step-end* marks in the *Undo* Stack before and after executing this operation. When our implementation of the *Undo* operator pops out this special *multi-step-end* mark, it realizes that it is undoing a multiple step operation and keeps executing undoes until it reaches the *multi-step-begin* mark. To place these special marks in the *Undo* stack we inserted a special piece of text in the document ("<beginning multiple-steps operation>") and then deleted it. Hence, we only left a footprint in the *Undo* stack.

3.1.2.2.2 Typing (and *AutoCorrect*) Replay

This example illustrates a case in which the Word API does not provide an exact method for replaying this operation. Although it provides a mechanism for inserting text into a document, this action does not always produce the same effect as when the text was originally typed through the GUI because the API operation does not trigger the *AutoCorrect* and *AutoFormat* rules built into Word.

We overcame this problem by implementing our own typing method that simulated the entry of this text through the GUI to Word as if a user was typing it. However, this technique had two undesirable side effects that required compensation. The first was a synchronization issue. When replaying the history of operations to reconstruct a corrupted document, a race condition was produced between the operations executed through the Word API and the typing operations fed through the simulated keyboard input. This race condition was remedied through synchronization primitives that sequentialized the two input streams.

The second side effect from replaying our typing operators was Word would split the input into different undoable chunks than it had originally. To compensate for this problem we employed a grouping mechanism similar to the one described in the implementation of the *Copy/Move* operation above

3.1.3 Ensuring Document Integrity

.

When a document is created the Integrity Manager creates a version history for it. Each time that a persistent copy of the document is saved, the Integrity Manager generates a universally unique ID to identify that version of the document and stores this ID within the document itself. It also adds a record to the document's version history that contains the unique version ID, the user that saved the document, a timestamp, the captured sequence of the changes performed to the document since the previous version, and a cryptographic integrity mark that is a function of the content of the document (i.e., a cryptographic checksum).

When a document is opened, the Integrity Manager finds its version record in the document's version history and checks the integrity of the document against the cryptographic integrity mark stored in the version history. If the document was corrupted or was modified outside of the control of the Integrity Manager then it will no longer match its cryptographic integrity mark. When this occurs the user is offered the option of reconstructing the corrupted document from its recorded change history.

3.1.4 Attribution

The availability of the history of changes performed to a document allowed us to implement a novel attribution scheme. It utilizes a *time lever* to allow the user to move forward or backward through the document's history while looking at its state at those points in time. At each instant, the system displays the attribution information associated with the current operation (i.e. who performed it and when the operation occurred). In this time lever tool, forward transitions are performed by executing the recorded sequence of changes while backward transitions are performed by executing Word's *Undo* command.

4. Discussion and Future Work

4.1 Managing the Complexity of Word

As noted in the Technical Challenges section, capturing all of changes to a document is the biggest challenge faced by this project. It is exacerbated by the size and complexity of the Word interface. Based on an analysis of Word's built-in tool for customizing its user interface there are over 1100 unique Word commands and almost 900 unique command bar controls. This includes a broad range of commands from those that change the text font to commands that move the insertion point, to print command, etc. Of these, roughly 270 affect the textual content of a Word document (i.e., excluding commands for drawing, manipulating forms, databases, pictures, webs, and tables).

However, most users only use a very small subset of these commands. Based on a survey conducted in our group, we found that only 19 of these commands were commonly used, an additional 42 were used infrequently, and the rest were hardly, if ever, used. We therefore focused our initial implementation effort on the commonly used subset and to a lesser extent the infrequently used subset. Currently, the Document Integrity Manager can capture and replay 89% of the commonly used commands, 19% of the infrequently used commands, and none of the rest.

4.1.1 Generic Capture and Replay

Given the formidability of the Word API, we recognized from the start of the project that individual capture and replay mechanisms could not be built for every Word command. We therefore devised a strategy for handling the remaining less frequently used commands through a generic capture and replay mechanism. This strategy relied upon the knowledge that almost all Word commands are scoped in their effect by the range of the current *Selection* (the few that didn't would have to be handled individually). Thus, if the state of this selection were cached before the execution of the modifying command, then the set of changes made by that command could be forensically discovered by comparing the cached state with the resulting state and extracting the differences.

This strategy also required finding a general mechanism for detecting that the document had been modified that was independent of the particular command that caused that change. Initially, we planned to use Word's *document-modified* attribute for detecting these generic changes, but this would have required us to continually reset it through the scripting API to *not-modified* so that we could detect when generic (i.e. otherwise undetected) changes had occurred and to restore it when Word actually needed to access this document attribute (i.e. for *Save* and *Auto-Save*).

We have since discovered an alternative generic change indicator that is better because it is more specific and because it only relies on monitoring the state of the document (rather than changing it as we would have had to do with the *document-modified* attribute). This alternative generic change indicator is the depth of Word's *Undo* stack. Each time the document is changed, Word pushes the change (in a proprietary inaccessible format) onto the *Undo* stack so that it can later be undone if the user so chooses. While we can't access the change itself from the *Undo* stack (a real pity – being able to do so would have greatly simplified the Capture challenge), we can detect the presence of the new item at the top of the stack, and that is sufficient to determine that the document has been modified.

We delayed the implementation of this generic capture and replay mechanism until now so that its implementation could utilize the mechanisms already built for the individual commands to extract changes from a completed command (by forensically comparing the cached state of the *Selection* with the resulting state) and the better understanding we have now developed of how Word functioned and how to utilize its scripting API (such as the recognition that the depth of the *Undo* stack was a better indicator of generic change).

4.2 Limitations

Although some operations are sensitive to Word's configuration (such as whether the *AutoCorrect* feature is turned on or not), we are not currently detecting and recording changes in this configuration. We are also not recording changes to the templates (including modifications to the *Styles*). Finally, although we might be able to detect the invocation of macros and invoke the same macro during document reconstruction, we wouldn't be able to reproduce the same behavior if that macro interacted with the environment (e.g., interactive input from the user or reading the contents of a file) unless those interactions were also captured.

4.3 Future Work

4.3.1 Enhanced Attribution

We will enhance our time lever tool so that it only displays changes that affect the user-selected portion of the document containing the change to be attributed. The time lever tool will extract from the history of changes the subset of operations that impacted that selected portion of the document and convert the coordinates of each of those extracted changes to be relative to the beginning of the selected portion of the document. To identify this subset of relevant changes, the tool will replay the modification history backwards keeping track of the selected range until it collapses. It will then replay the change history forward keeping track of the selected range and extracting the operations that impact that range. Finally, the time lever tool will interactively display this condensed history –containing only the extracted operations that affect the selected portion of the document – so that the user can zero in on the particular change requiring attribution.

4.3.2 Additional Document Integrity Managers

After completing the implementation of the Word Document Integrity Manager, we plan to create a Document Integrity Manager for PowerPoint so that briefings could also be protected from malicious attacks and changes to those briefings could be attributed. We expect to leverage the current implementation, particularly the generic capture and replay mechanism.

We also plan to simplify the development of future Integrity Managers so that this technology can be applied to a broader set of COTS tools by developing a generic Integrity Manager that performs all of the common functions (such as recording the captured modifications, maintaining the document's version history, attaching digital signatures to the document, detecting that a document is corrupted when it is loaded, and providing attribution services), and interfaces to COTS-specific modules that capture the modifications in that COTS tool and replay recorded changes during document reconstruction.

We also plan to examine other applications of this technology, such as Instructional Tutoring systems, that take advantage of its capability to monitor user actions in application-specific terms.

5. References

[1] Robert Balzer and Neil Goldman: Mediating Connectors: A Non-ByPassable Process Wrapping Technology, DARPA DISCEX Conference 2000, Hilton Head SC, Jan 25-27, Vol II, pp 361-368.

[2] Neil Goldman and Robert Balzer: The ISI Visual Design Editor Generator, 1999 IEEE Symposium of Visual Languages, Tokyo, Japan, Sept 13-16, pp 20-27.